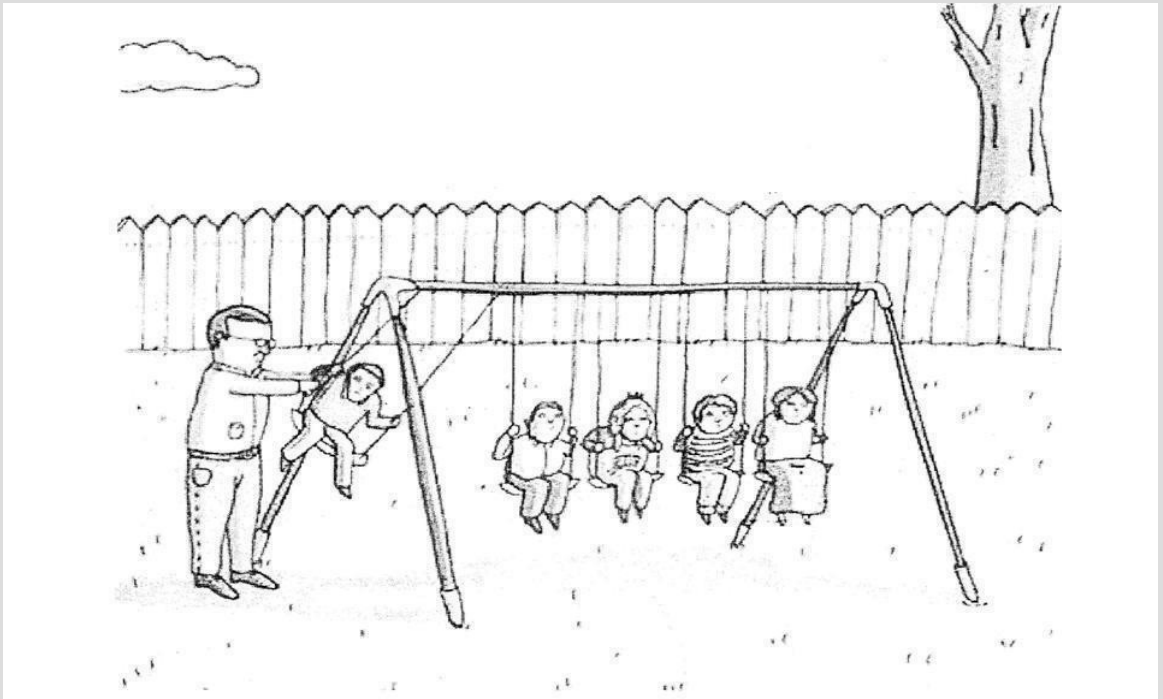
Arrays (& strings)

Ch 7



Why science teachers are not asked to monitor recess.

Highlights

- arrays

```
int x[4];
x[0] = 1;
```

- 2D arrays

```
int box[3][4];
// 3 rows, 4 columns
```

- string functions

```
string x = "hello there!";
cout << x.substr(x.find('t'));
```

- arrays and functions

```
int x[3];
foo(x);
```

string

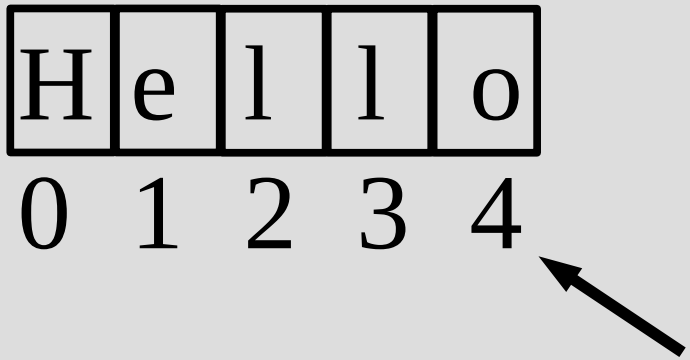
We have been using strings to store words or sentences for a while now

However, when we type “string x” it does not turn blue, as it is not a fundamental type (like char)

strings are basically a grouping of multiple chars together in a single variable

string index

String greeting = “Hello”;

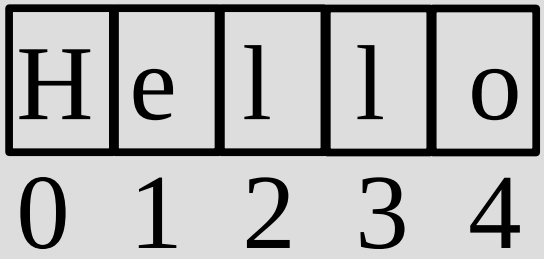


The position of a character is called its index.

Note that the index starts from zero, not one (this is just to make your life miserable)

string functions

String greeting = “Hello”;

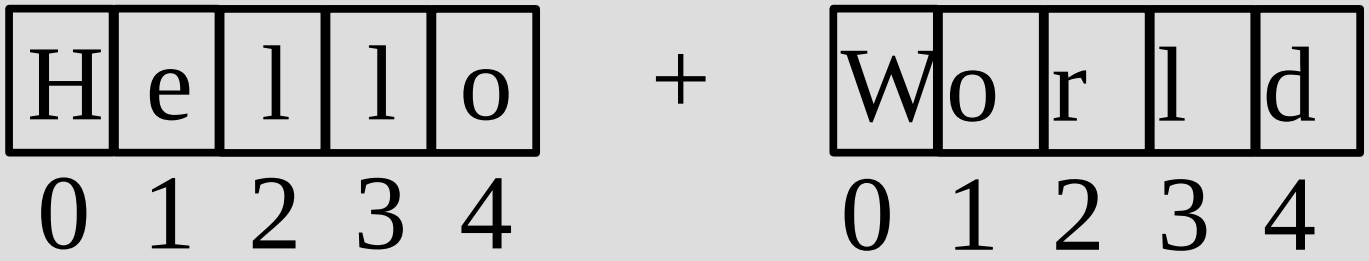


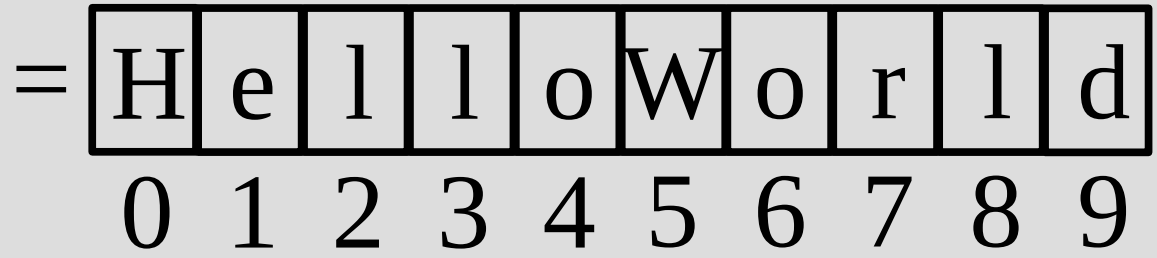
greeting.length();

→ returns value 5 (int)

Tells how many characters are in the variable

string concatenation





String concatenation does not automatically add a space (see: stringConcatenation.cpp)

strings

There are also some other useful functions (see book or google for a full list)

Some of the more useful ones are:

- .at(int index): character at the index
- .find(): finds first character or string
- .substr(int start): pulls out part of the original string

(see: string.cpp)

C-Strings and strings

There are actually two types of “strings” (multiple characters) in C++

A C-String is a char array, and this is what you get when you put quotes around words

```
cout << "HI!\n";
```

A string (the thing you #include) is a more complicated type called a class (few weeks)

C-Strings and strings

It is fairly easy to convert between C-Strings and strings:

```
char cString[] = "move zig";
string IMAstring = cString;
cout << IMAstring.c_str() << endl;
// above converts it back to C-String
```

You can also convert between numbers and strings:

```
char number1[20];
string number2;
cin >> number1 >> number2;
cout << "sum is: " << (atof(number1) + stod(number2)) << endl;
```

(see: stringConversion.cpp)

C-Strings and strings

C-Strings are basically strings without the added functions

```
char word[] = {'o', 'm', 'g', '\0'};
```

You should end C-Strings with null character, as this tells cout when to stop displaying

This means you can initialize char arrays with quotes (**BUT NOT OTHER ARRAYS**) (see: cstring.cpp)

Arrays

Arrays are convenient ways to store similar data types (like multiple chars for a string)

Arrays are indexed starting from 0,
so index 0 is the first element,
index 1 is the second element ...

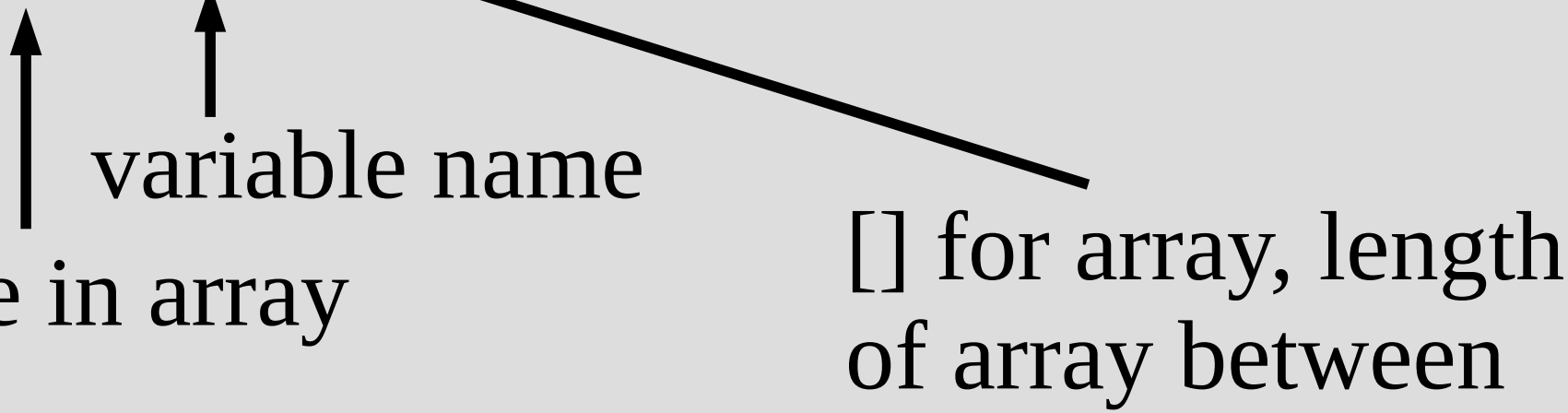
Unlike strings, you can make an array of whatever type you want (any type!)

Arrays - declaration

When making an array, you need both a type and a length

The format for making an array is below:

```
int x[5]; // 5 ints
```



Arrays - elements

To access an element of an array, use the variable name followed by the index in [ ]

```
x[1] = 2;
```

(See: simpleArray.cpp)

Arrays

Note that the number in the [ ] is inconsistent:

1. First time (declaration): this is the length

2. All other times: this is the index of a single value inside the array

If you want to indicate a whole array, just use the variable name without any [ ]
(more on this later)

Arrays - manual initialization

Arrays can be initialized by the following: (must be done on declaration line!)

```
int x[] = {1, 4, 5, 2};
```

If you access outside of your array you will either crash or get a random value

You can also use a constant variable to set the size:
(See: average.cpp)

```
const int size = 8;
int x[size];
```

Arrays

When you make an array, the computer reserves space in memory for the size

The array variable is then just a reference to the first element's memory location

The computer simply converts the index into an offset from this initial location (see arrayAddress.cpp)

Memory

Memory:

Code:

Memory (declaration)

Memory:

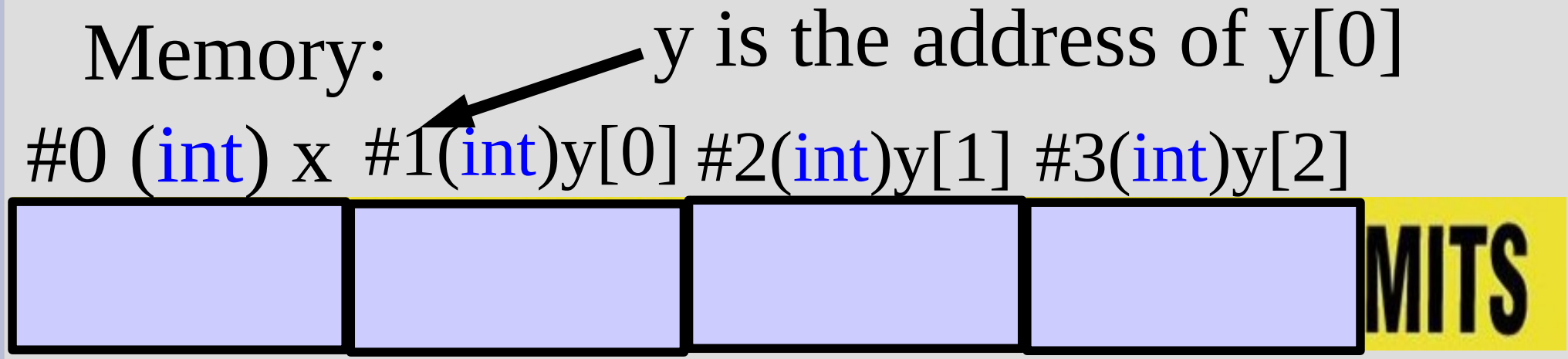
#0 (int) x

Code:

```
int x;
```

Memory (declaration)

Memory: y is the address of y[0]



Code:

```
int x;
int y[3];
```

Arrays - looping

As arrays store multiple elements, we very often loop over those element

There is a special loop that goes over all elements (for each):

```
int x[] = {1, 4, 5, 2};

for(int a : x)
{
```

x is an array

a has the value of x[i] for each i

(See: forEach.cpp)

Partially filled arrays

Arrays are annoying since you cannot change their size

You can get around this by making the array much larger than you need

If you do this you need to keep track of how much of the array you are actually using

(See: partiallyFilled.cpp)


["Hip" , "Hip"]
Hip Hip Array

Array - element passing

Each element of an array is the same as an object of that type

For example: `int[] x = {1, 2};`
x[0] is an int, and we can use it identical as if we said: `int x0 = 1;`

(See: maxPassInt.cpp)

Array - array passing

Arrays are references (memory addresses)

This means we can pass the reference as an argument in a method

Then the method can see the whole array, but it won't know the size

(See: maxPassArray.cpp)

Array - array passing

But wait! This means the function can change the data since we share the memory address

(See: reverse.cpp)

Array - array passing

If we want to prevent a function from modifying an array, we can use const in the function header:

```
void reverse(const int word[]);
```

This also means any function called inside reverse must also use const on this array

(See: reverseFail.cpp)

Array - returning arrays

However, we do not know how to return arrays from functions (yet)

```
int[] foo(){
    int x[] = {1,2};
    return x;
} // x dies here, what are you returning?
```

syntax error

For now, you will have to pass in an array to be changed, much like call-by-reference

Sort

Let's practice arrays by sorting!

(See: sort.cpp)

Sort

Let's practice arrays by sorting!

Plan of attack:

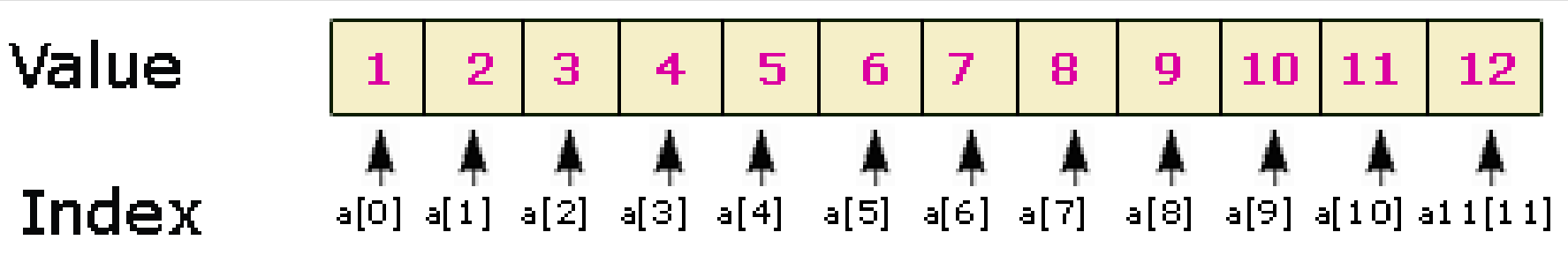
1. Make a new array
2. Find minimum element in original array and copy into new array
3. Replace minimum element in original array with the maximum element
4. Repeat 2 to 3 until done

(See: sort.cpp)

Multidimensional Arrays

So far we have dealt with simple (one dimensional) arrays

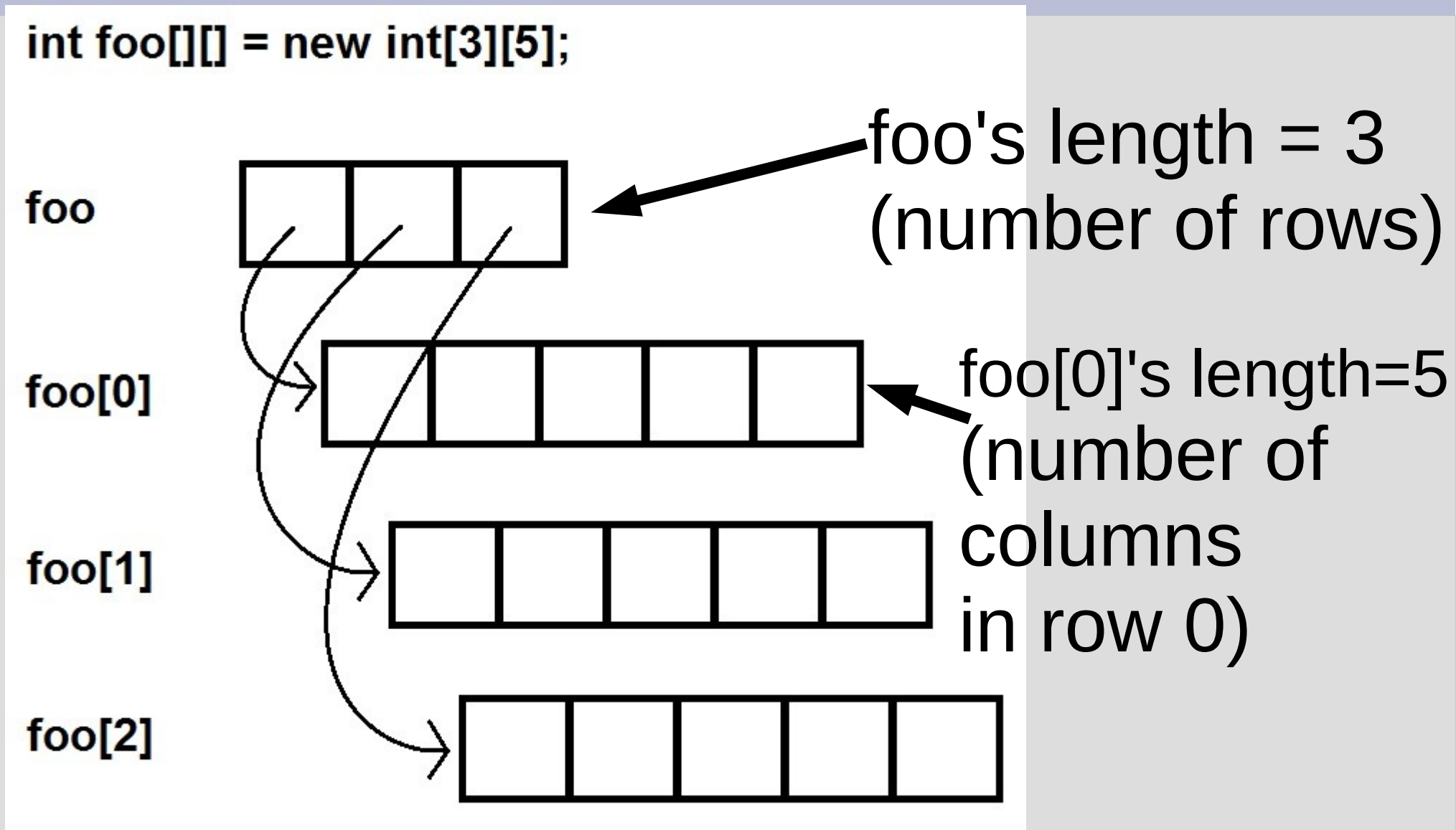
We have represented this as all the data being stored in a line



(See: lineWorld.cpp)

Multidimensional Arrays

```
int foo[][] = new int[3][5];
```



Multidimensional Arrays

If we think of a couple simple (one dimensional) arrays on top of each other...

Index	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

One array for numbers 1-10

One array for numbers 71-80

(See: gridWorld.cpp)

Multidimensional Arrays

Recreate:

Index	0	1	2	3	4	5	6	7	8	9
0	1	2	3	4	5	6	7	8	9	10
1	11	12	13	14	15	16	17	18	19	20
2	21	22	23	24	25	26	27	28	29	30
3	31	32	33	34	35	36	37	38	39	40
4	41	42	43	44	45	46	47	48	49	50
5	51	52	53	54	55	56	57	58	59	60
6	61	62	63	64	65	66	67	68	69	70
7	71	72	73	74	75	76	77	78	79	80
8	81	82	83	84	85	86	87	88	89	90
9	91	92	93	94	95	96	97	98	99	100

(See: oneToAHundred.cpp)